A APPENDIX

In the appendix, we have included comprehensive guidelines for artifact evaluators on how to install and execute Pantheon. The source codes of Pantheon can be accessed at https://github.com/PantheonInfer/Pantheon. Furthermore, additional resources, such as the datasets used and trained model files, can also be obtained through the link.

A.1 Prerequisites

In this subsection, we detail the hardware and software requirements for both our offline and online modules. The offline module handles DNN slicing and variant generation, while the online module includes the runtime and profiling parts. Additionally, we describe the necessary Jetson board device settings that need to be configured prior to evaluation.

Offline modules. Offline modules operate on a desktop (with Windows, although others may also be compatible) connected to the Jetson board with an Ethernet cable. The main software requirements for the offline modules are as follows:

- *Python 3.8*
- *PyTorch 2.0.1*
- *PyTorch Lightning 1.1.0*
- *pymoo 0.6*

For the complete list of required Python packages, you can refer to the `requirements.txt` file included in the released source codes and install them via:

```
pip install -r requirements.txt
```

Online modules. To conduct testing on the online modules, it is necessary to have an NVIDIA Jetson Xavier NX Developer Kit with Jetpack 5.1.1 installed. The following are the software requirements and installation steps for the online modules:

CMake. It is typically installed by default. Please ensure that the version installed is equal to or newer than 3.17.5.

LibTorch. It will be available once PyTorch is installed successfully on Jetson boards. For installing PyTorch on Jetson, please refer to NVIDIA official guidelines.

OpenCV. It is usually installed by default and can be checked by

```
cd /usr/bin
./opencv_version
```

Protobuf. It should be installed via the following command:

```
sudo apt-get install protobuf-compiler
↪  libprotobuf-dev
```

spdlog. It should be installed via the following commands:

```
git clone https://github.com/gabime/spdlog.git
cd spdlog && mkdir build && cd build
cmake .. && make -j
make install
```

Device settings. The Jetson Xavier NX board is powered by a 19V 2.37A 45W AC adapter and connected to the desktop using an Ethernet cable. The NVP mode of the Jetson board should be set to `MODE_20W_6CORE` by:

```
sudo nvpmodel -m 8
```

The dynamic voltage frequency scaling governor should be disabled, and the processors' clocks should be locked to their maximums using:

```
sudo jetson_clocks
```

A.2 Usages

We provide the pre-trained DNN models and the required datasets, both of which can be downloaded from the provided link. Once the pre-trained models and datasets are downloaded, you can run Pantheon by following these steps:

1) DNN slicing and variant generation. In this step, we slice a given pre-trained DNN into chunks, insert early exits in between all chunks, and then train all inserted early exits. These can be done by simply running the script `offline/construct.py` with the arguments listed in Table 2.

Argument	Help
–task	Specify the DNN task.
–data_dir	The root directory of all datasets.
–log	The directory to store training logs.
–learning_rate	Learning rate for training early exits.
–batch_size	Batch size for training early exits.
–weight_decay	Weight decay for learning rate.
–max_epochs	Number of epochs for training early exits.
–num_workers	Number of threads for loading datasets.
–gpu_id	Specify the GPU for training in case multiples are available.
–pretrain	The path to the pre-trained model weights.

Table 2: Arguments of `offline/construct.py`.

For example, to slice and generate variants for the traffic sign classification task, we can run the following script:

```
python offline/construct.py --task
↪  sign_recognition --data_dir /path/to/datasets
↪  --log /path/to/logs --max_epochs 100
↪  --weight_decay 1e-3 --pretrain
↪  /path/to/weights
```

For the argument values for each task, please refer to Table 3.

As the training process takes a long time, we also provide the trained variants for all tasks through the link for downloading.

2) Model format transformation. The generated model variants are now saved in PyTorch style, which is not available to LibTorch (C++

Task name	–task	–learning_rate	–batch_size	–weight_decay	–max_epochs
Traffic light detection	object_detection	1e-3	24	5e-4	50
Traffic sign classification	sign_recognition	1e-2	256	1e-3	100
Face Detection	face_detection	1e-3	24	5e-4	100
Age Classification	age_classification	1e-3	128	1e-4	200
Gender Classification	gender_classification	1e-3	128	1e-4	50
Emotion Recognition	emotion_classification	1e-2	64	5e-4	100
Wildfire Detection	wildfire_detection	1e-3	24	5e-4	100
Wildlife Identification	wildlife_recognition	1e-2	64	5e-3	100
Scene Recognition	scene_recognition	1e-2	256	1e-2	100

Table 3: `offline/construct.py` argument values for tasks.

front-end). Hence, the next step is to export the generated model variants to JIT format by running the script `offline/export.py` with the arguments listed in Table 4.

Argument	Help
–task	Specify the DNN task.
–save	The directory to save the exported model weights.
–weights	The path to the generated variants.

Table 4: Arguments of `offline/construct.py`.

We also provide all exported variant weights in JIT format, which can be accessed through the provided link.

3) Profiling. In this step, we profile the accuracy, latency, and memory costs of model variants. To profile the latency on the Jetson board, we should first compile the profiler on it as follows:

```
cd online/apps/profiler
mkdir build
cd build
cmake -DCMAKE_PREFIX_PATH=`python3 -c 'import
↪  torch;print(torch.utils.cmake_prefix_path)'`
↪  ..
cmake --build . --config Release
```

Then, we need to run the profiler on the Jetson board by:

```
./profiler
```

At the same time, we should run the script `offline/_profile.py` on the desktop with arguments listed in Table 5. This script with communicate with the profiler to evaluate the latency on board, evaluate the memory costs and accuracy on the desktop, and generate the profiling results as a CSV file automatically.

Argument	Help
–task	Specify the DNN task.
–data_dir	The root directory of all datasets.
–weights	The directory holding exported variants.
–host	IP for TCP/IP connection with the device (default value will do).
–port	Port for TCP/IP connection with the device (default value will do).
–buffer	Buffer size for TCP/IP connection with the device (default value will do).
–batch_size	Batch size for evaluating accuracy.
–num_workers	Number of threads for loading datasets.

Table 5: Arguments of `offline/construct.py`.

For example, we can simply run it like this:

```
python offline/_profile.py --task sign_recognition
↪  --weights path/to/weights
```

4) Early exit placement optimization. With the profiled results, in this step, we optimize the placement of early exits and remove redundant exits for actual deployment. This is done for all tasks in the same application at once by running the script `offline/plan.py`:

```
python offline/plan.py --base
↪  path/to/variants/weights/of/all/tasks
↪  --setting path/to/application/settings --save
↪  path/to/save/variants/for/deployment
```

The `–base` argument should be the directory holding the variants' weights in JIT format for all tasks in the specific application. The `–setting` specifies the constraint for that application, including the allowed maximum accuracy drop for each task and the memory constraints, which are given as a JSON file. We provide the settings for all applications in `experiments/settings/deploy`. The `–save` argument specifies the directory to hold the model variants for deployment, which should be uploaded to the Jetson board.

5) Deployment. The last step is to deploy the application to the Jetson board. First, we should compile the runtime on board:

```
cd online/apps/runtime
mkdir build
cd build
cmake -DCMAKE_PREFIX_PATH=`python3 -c 'import
↪  torch;print(torch.utils.cmake_prefix_path)'`
↪  ..
cmake --build . --config Release
```

Then, we start Pantheon runtime to execute the application with specified workloads

```
./runtime /path/to/models /path/to/workload
↪  /path/to/logs
```

The first argument is the path to the directory storing the variants generated in the previous step. The second argument specifies the workload, which is given in Protobuf format. We provide the workloads for each application in `experiments/settings/workload`. The last argument specifies the path to store the runtime logs, which are then used to evaluate runtime performance.

At the end, we can download the runtime logs to the desktop and run `experiments/logs/scheduling/viz.py` to evaluate the runtime performance:

```
python experiments/logs/scheduling/viz.py
↪  /path/to/logs /path/to/workload
```